UNIVERSITY OF ARIZONA

Topics in Low-Level Reverse Engineering, with Applications to Software Security

Final Project Report

Contract No.: FA9550-07-1-0019
Principal Investigator: Saumya Debray
Period of Contract: Dec 1, 2006 to Nov 30, 2009
Date of Report: Jan 27, 2010

20120918194

Final Performance Report

Project Information

Contract No.: FA9550-07-1-0019

Contract Title: **Topics in Low-Level Reverse Engineering, with Applications to Software Security**

Principal Investigator: Saumya Debray

Period of contract: Dec 1, 2006 to Nov 30, 2009

Date of report: Jan 27, 2010

Project Objectives

Malware attacks on computer systems have increased sharply in recent years. Mitigating the effects of such attacks requires quick response. However, this is hampered by the many layers of anti-analysis defenses mounted by malware code, which existing program analysis techniques do not handle well and which therefore require time-consuming and tedious manual intervention. The overall goal of this project was to develop techniques to automate the analysis of malware executables, and thereby accelerate the process of identifying the internal logic of the malware code. A secondary goal was to develop automated techniques to identify and—where possible, eliminate—anti-analysis defenses in the code, so as to simplify subsequent analysis.

Significant Accomplishments

The significant accomplishments of the project consist of the following:

1. [*Foundations*]: We developed a formal semantic model for self-modifying code.
2. [*Static Extraction of Malware Code*]: We developed a technique to use program analysis algorithms to identify the decryptor routine in a malware binary, and then modify this decryptor routine in such a way that it can be used to extract the malware code.
3. [*Anti-analysis Defense Detection*]: We developed a technique to identify dynamic anti-analysis defenses in the malware code by analyzing the control-flow structure of the code leading to the decryptor routine.

4. [*Detection of Disassembly Errors*]: We developed a machine-learning-based technique to identify code regions in a disassembled file that appear to contain disassembly errors.

We discuss these items in more detail below.

1. Foundations

Automated analysis of obfuscated malware code requires the application of program analysis algorithms. However, classical program analysis algorithms, which are used in the areas of compilers and software engineering, presuppose that the code being analyzed is immutable. They are therefore inapplicable to self-modifying programs, which includes most malware. To address this problem, we formulated a formal semantics for self-modifying code [DCT08]. To the best of our knowledge this is the only work on formally modeling self-modifying code that makes it straightforward to identify code regions where classical program analysis algorithms can be applied with the appropriate soundness guarantees. It forms the formal basis for our work on Static Code Extraction and Anti-Analysis Defense Identification.

A detailed description of the formal semantics is given in the publication referenced below [DCT08]. It is available from the PI's web page (www.cs.arizona.edu/~debray/Publications).

2. Static Extraction of Malware Code

Most malware code is transmitted in encrypted or "packed" form and unpacked at runtime prior to execution. In many cases, the unpacking routine that restores the code to its original executable form is guarded by various kinds of defensive checks aimed at making it harder to reverse-engineer the code. For example, such defenses might cause the unpacker to be invoked—and the malware payload to be exposed for analysis—only on specific dates ("time bombs") or when executed in a specific environment ("logic bombs"). The objective of this part of the project was to devise techniques to extract the code via static analysis, thereby sidestepping such dynamic defenses.

We developed an algorithm to statically extract packed malware code [CDKT09]. There are three conceptual parts to this algorithm. First, memory analysis is used to identify memory locations that are modified, and program slicing is then used to identify the unpacker code. The second part then uses the control flow structure of the code to identify—and, where possible, eliminate—runtime anti-analysis defenses. Finally, the unpacker code is emulated in a sandboxed environment to extract the malware code.

We implemented our ideas in a binary analysis tool consisting of roughly 81,000 lines of C code. This system was evaluated on a variety of packed malware code that used both commercial packers, such as UPX and tElock, as well as a number of different custom-crafted packers. In each case, we used a manual analysis to determine the actual unpacked instruction sequence, and compared this with the unpacked code obtained using our analysis tool to determine the extent to which our tool was able to successfully extract the packed malware code.

Our evaluation of this algorithm showed that it could handle a variety of commercial as well as custom unpackers. One engineering issue that posed a hurdle was that of replicating OS-level features with sufficient fidelity within the sandboxed unpacker: this is necessary to deal with unpacker code that makes system calls as part of the unpacking process. This, in turn, limited the extent to which our implemenation was able to handle multi-level unpacking.

A detailed description of this algorithm and our evaluation data are available in the publication referenced below [CDKT09]. It is available from the PI's web page (www.cs.arizona.edu/~debray/Publications).

3. Anti-Analysis Defense Identification

As discussed above, one of the steps in the static extraction of malware code is to identify the unpacker code using program analysis techniques. These same program analyses also reveal the control flow structure of the code leading up to the unpacker. An examination of this control flow structure then makes it possible to detect the possible presence and nature of anti-analysis defenses in the code [CDKT09]. Furthermore, the analysis of this code reveals whether there are any dependencies between the defense code and the unpacker code—if, as is usually the case, there are no dependencies, the defense code can be removed without affecting the unpacker code. This therefore provides an approach to the identification and elimination of runtime anti-analysis defenses with semantic soundness guarantees.

A detailed description of this algorithm and our evaluation data are available in the publication referenced below [CDKT09]. It is available from the PI's web page (www.cs.arizona.edu/~debray/Publications).

4. Detection of Disassembly Errors

A fundamental assumption made by all existing approaches for static analysis of malware executables (including ours) is that the malware code has been accurately disassembled. This assumption may not hold true if the malware being analyzed uses anti-disassembly defenses, i.e., techniques intended to introduce errors into disassembled code. Moreover, the ubiquitous Intel IA-32 architecture has a very high density of instruction encoding, i.e., almost any sequence of bytes decodes to some legal instruction—it is unusual for a disassembly (even one that contains numerous errors) to actually encounter an illegal byte sequence. This means that disassembly errors usually occur silently, substituting erroneous instructions for the correct ones without any external indication that something has gone wrong. The reason this is an issue, even using current dynamic analysis (which observe the execution of the program being analyzed and therefore are presumably immune to anti-disassembly defenses), is that dynamic analyses can only examine those parts of the program that were executed; the remainder—which may contain conditionally unpacked code—has to be examined using the results of static disassembly. If the static disassembly contains undetected errors, subsequent analyses based on the incorrect disassembly propagate errors silently up the entire analysis chain. Our objective in this portion of the project was to develop techniques to analyze an instruction sequence obtained via static disassembly and identify possible errors.

We carefully examined a large body of correct and incorrect disassemblies and observed that the erroneous disassemblies, while technically "legal", looked different from correct disassemblies. Sometimes the discrepancies were quite obvious, e.g., addresses that did not correspond to any location within the program; at other times they were subtle, e.g., the use of unusual combinations of instructions or addressing modes. We then used machine learning techniques to systematize the process of distinguishing correctly disassembled code from incorrect disassemblies. The idea is to use a *training set* of examples of both correct and incorrect disassemblies, and use machine learning to determine which combinations of features of these disassemblies correspond to correct disassemblies and which do not. The resulting classifier can then be used to detect possible errors in other disassembled code. The accuracy of this classifier depends on the extent of coverage provided by the training set. We evaluated our classifier on a number of disassemblies obtained from obfuscated binaries and found that, with a little care in selecting the training input, it is possible to obtain quite accurate error classification in general [KDF09].

A detailed description of this algorithm and our evaluation data are available in the publication referenced below [KDF09]. It is available from the PI's web page (www.cs.arizona.edu/~debray/Publications).

Theses and Dissertations

In addition to the publications mentioned above, the research effort has led to one MS thesis [Krishnamoorthy] and one PhD dissertation [Coogan]. We describe their most significant contributions below.

1. Kevin Coogan,"Automatic Deobfuscation of Malware Executables."

Doctoral Dissertation, The University of Arizona, Tucson.

Expected Completion Date: December 2010.

Executive Summary

Computer malware typically resort to a variety of techniques to make it difficult for others to understand the internal logic of the malware code; these techniques are usually referred to as "code obfuscations." The effect of such obfuscations is to slow down the process of understanding the malware and devising countermeasures. This dissertation investigates techniques to automate the process of identifying and eliminating the effects of obfuscation, and thereby extracting the essential internal logic of the malware code, with the intent of simplifying and speeding up the task of developing countermeasures to new malware. Two different approaches are explored: *static analysis*, where the malware sample is analyzed without running it; and *dynamic analysis*, where the malware is executed in a suitably isolated environment and its execution observed. In the case of static analysis, the dissertation describes a technique to use program analysis techniques to identify code that the malware would decrypt when executed, and extract this code automatically without running the malware. In the case of dynamic analysis, the dissertation develops techniques to examine the sequence of instructions executed by the malware sample and identifying those instructions that are irrelevant to its observable behavior and which can therefore be discarded, thereby reducing the set of instructions that have to be considered when understanding the malware code.

Note: The dissertation is expected to be completed by December 2010. At that time it will be available from the web site of the University of Arizona Department of Computer Science, or via email from the PI (email: debray@cs.arizona.edu).

People involved in the research (in addition to the PI): Kevin Coogan, Gregg Townsend, TasneemKaochar, Amr Gaber.

Publications resulting from this research:

1. S. K. Debray, K. P. Coogan, and G. M. Townsend. On the Semantics of Self-Unpacking Malware Code. Technical Report, Dept. of Computer Science, University of Arizona, Tucson. July 2008. http://www.cs.arizona.edu/~debray/Publications/self-modifying-pgm-semantics.pdf

2. Kevin Coogan, Saumya Debray, Tasneem Kaochar, andGregg Townsend. Automatic Static Unpacking of Malware Binaries. *Proc. 16th. IEEE Working Conference on Reverse Engineering*, October 2009, pp. 167-176.

2. Nithya Krishnamoorthy, "Automatic Static Detection of Disassembly Errors"

MS Thesis, The University of Arizona, Tucson

Expected Completion Date: May 2010

Executive Summary

When someone wants to understand the internal logic of a malware program, the first step is to take the malware file, which is in a format suitable for execution on a computer, and extract from it a human-readable representation of the machine instructions in the malware code. This process is known as disassembly, and all of the work on malware analysis assumes that the disassembly is successful and that the human-readable representation obtained is correct. Unfortunately, in practice disassemblies may contain errors, and the errors may not always be detected by the disassemble. When this happens, the conclusions drawn from subsequent analyses of the erroneous disassembly are also wrong.

This thesis describes an approach for automatically detecting locations within a disassembly that may contain errors. The key insight in this work is that when a disassembly error occurs, the resulting (erroneous) instruction sequence is subtly different from disassemblies that are correct. The thesis proposes an approach that uses machine learning techniques to "learn" how to distinguish between a sample set of correct disassemblies from a sample set of incorrect disassemblies. This results in a software tool that can be used to detect incorrect disassemblies. Experimental studies indicate that the resulting tool can automatically detect errors in disassemblies with a high degree of precision.

Note: The thesis is expected to be completed by May 2010. At that time it will be available from the web site of the University of Arizona Department of Computer Science, or via email from the PI (email: debray@cs.arizona.edu).

People involved with this research (in addition to the PI): Nithya Krishnamoorthy, Keith Fligg.

Publications resulting from this research:

1. Nithya Krishnamoorthy, Saumya Debray, and Keith Fligg. Static Detection of Disassembly Errors. *Proc. 16th. IEEE Working Conference on Reverse Engineering*, October 2009, pp. 259-268. (A revised and extended version of the paper has been invited to a special issue of the journal *Science of Computer Programming*.)

Software Resulting from the Research Effort

The research effort has led to the development of prototype software tools used for validation and evaluation of the research. These consist of the following:

1. A tool for automatic code extraction from packed malware executables. This consists of roughly 81,000 lines of C code. A README file for this system is given in Appendix 1. The software is available from the PI via email.
2. A tool for automatic detection of disassembly errors. This consists of a total of roughly 65,000 lines of C code. A README file for this system is given in Appendix 2. The software is available from the PI via email.

References

[Coogan]

Kevin Coogan. Automatic Deobfuscation of Malware Executables. Doctoral dissertation, University of Arizona. Dec. 2010 (expected).

[DCT08]

S. K. Debray, K. P. Coogan, and G. M. Townsend. On the Semantics of Self-Unpacking Malware Code. Technical Report, Dept. of Computer Science, University of Arizona, Tucson. July 2008. http://www.cs.arizona.edu/~debray/Publications/self-modifying-pgm-semantics.pdf

[CDKT09]

Kevin Coogan, Saumya Debray, Tasneem Kaochar, andGregg Townsend. Automatic Static Unpacking of Malware Binaries. *Proc. 16th. IEEE Working Conference on Reverse Engineering*, October 2009, pp. 167-176.

[Krishnamoorthy]

Nithya Krishnamoorthy. Automatic Static Detection of Disassembly Errors. MS Thesis, Dept. of Computer Science, The University of Arizona, May 2010 (expected).

[KDF09]

Nithya Krishnamoorthy, Saumya Debray, and Keith Fligg. Static Detection of Disassembly Errors. *Proc. 16th. IEEE Working Conference on Reverse Engineering*, October 2009, pp. 259-268. (A revised and extended version of the paper has been invited to a special issue of the journal *Science of Computer Programming*.)

APPENDIX 1. Online README file for Automatic Static Decryption Tool

The tool described earlier for satic extraction of malware code consists of approximately 81,000 lines of C code. It is not practical to attach this to this report; instead we attach the README file describing this tool. The software itself can be obtained from the PI (email: debray@cs.arizona.edu).

```
1.0  System Requirements and Build
------------------------
The system is meant to run on linux 2.6.28-13 or higher.
The Boehm-Weiser garbage collector should be installed on your system.

To build, type make all from top level install directory.  Note that there
is no make install option.  Executable file is located in
<install_dir>/bin.

2.0 Executing sytem
-------------------
System expects an executable file name as input.  To see a full list of
options, type bin/plto with no parameters.

To run the code on a particular executable file, use:
/home/me/brevengg/> bin/plto <executable_name>

for example
/home/me/brevengg/> bin/plto sample/Hybris.c_defense1.exe

Code will produce binary file if transition point is found, and it matches
one
of the potential transition points.  This binary file is the dump of
memory
for the code at the point of tranistion to unpacked code.

3.0 Source file listing
-----------------------
<install_dir>/:
Makedefs
Makefile
README

<install_dir>/sample:
Hybris.c_defense1.exe   //sample malware file to test code
                        //WARNING -- THIS IS A LIVE WINDOWS
VIRUS!!!!!!!!!!!!!

<install_dir>/src:
addr_translation.c      //function to translate virtual addrs to runtime
addrs
aloc-list.c             //aloc data structures -- pointer set analysis
```

```
aloc-list.h
bbl.c                       //basic block data structure functions
cache.analyze.c             //instruction cache (performance increase)
cache.analyze.h
call-analysis.c             //analyze function calls and dependencies
call-analysis.h
clone.c                     //copy various data types
clone_small.c               //DEPRACATED
config.c                    //system initialization
cp.c                        //constant propogator
decrypt.c                   //build and execute unpacker (decryptor)
deobfuscate.c               //remove zero effect code
d_graph.c                   //visual graph generation
disassemble.c               //disassemble executable's byte code
disassemble.h
dominator.c                 //pre/post- dominator analyses
ep.c                        //entry point calculation and handling
flowgraph.c                 //build and manipulate control flow graph
function.c                  //build and manipulate function information
hash.c                      //instruction hashing data structure
instr.c                     //instruction data structure functions
kreugel-disasm.c            //disassembler built on Kruegel's paper
kreugel-junk.c
linkage.c                   //functions for adjusting links on basic blocks
liveness.c                  //liveness analysis functions
liveness.ci.c
liveness.ci.h
liveness.cs.c
liveness.cs.h
loop-analysis.c             //find loops
loop-analysis.h
lto.c                       //link time optimization (LEGACY CODE NOT USED)
lto.h
main.c                      //main
Makefile
misc.c                      //functions that don't fit anywhere else
NDG.c                       //non-directed graph functions
opt.h                       //optimization functions
opt.inline.c
opt.memopt.c
opt.peephole.c
opt.unreachable.c
phase.c                     //phase data structure functions
plto.c                      //drives code, where most of top level work is
done.
plto.h
potential.c                 //evaluation of different techniques, internal
only
potential.h
print.c                     //various print routines
print.h
relocate.c                  //handles relocations (LEGACY CODE NOT USED)
slice.c                     //calculates backward static slice
```

```
sloop.c                  //generates text for sloop graphics program
stack.analyze.c          //functions for various stack analyses
sysdep.c                 //handles system dependencies
transition-points.c      //calculates and tracks potential transition
points
win_wrappers.c           //enables calling of windows functions if on
cygwin

<install_dir>/aenv:
                         //abstract environment data structures and related

absenv.c
absint.h
aloc.c
alocenv.c
bool3.c
flags.c
Makefile
memregion.c
strditvl.c
valueset.c

<install_dir>/arch:
elf
i386

<install_dir>/bin:
README

<install_dir>/doc:
                         //some documentation related to source code
                         //not made by make all at top level.
                         //cd into dir and make
bitmanual
code_docs
CodingStyle.txt
Makefile
man
programmer_manual
usermanual
wbt01

<install_dir>/elf:
                         //code related to elf specific arch (LEGACY CODE)
constants.h
file.h
Makefile
print.c
process-binary.c
protos.h
read.c
write.c

<install_dir>/include:
```

```
addr_translation.h
aenv.h
aloc-list.h
basetsd.h
bbl.h
clone.h
config.h
cp.h
CVS
decrypt.h
deobfuscate.h
d_graph.h
dominator.h
edge.h
edge.type.h
elf.h
elfio.h
ep.h
file.h
flags.h
flist.h
flowgraph.h
function.h
global.h
hash.h
instr.h
kreugel-disasm.h
linkage.h
liveness.h
macros.h
NDG.h
operand.h
opt.layout.h
opt.unreachable.h
phase.h
plto_types.h
ptr_set_analysis.h
range.h
relocate.h
relocation.h
sdkddkver.h
section.h
slice.h
sloop.h
stack.analyze.h
string_table.h
symbol.h
sysdep.h
transition-points.h
util.h
utilities.h
valprof.h
windef.h
```

```
winerror.h
winnt.h
win_wrappers.h
x86-asm.h
x86-classes.h
x86.h
x86-liveness.h
x86-registers.h
x86-table-defs.h
x86-util.h

<install_dir>/lib:
Makefile

<install_dir>/libtest:
                        //some data structure test packages. not made
                        //by make all at top level.
                        //cd into dir and make
aloc.c
alocenv.c
alocenv.std
aloc.std
appmap.c
appmap.std
arraylist.c
arraylist.std
bool3a.c
bool3a.std
bool3b.c
bool3b.std
check
collector.c
collector.std
flags.c
flags.std
hacks.c
hashtable.c
hashtable.std
linkedlist.c
linkedlist.std
Makefile
memregion.c
memregion.std
README
strditvl.c
strditvl.std
valueset1.c
valueset1.std
valueset2.c
valueset2.std

<install_dir>/old-windisasm:
                        //old version of windiasm files (NOT USED)
```

```
basetsd.h
bbl.h
cmdline.c
dis.c
error.c
error.h
function.h
global.h
hashtable.c
helper.c
instr.h
iprint.c
linkedlist.c
main.c
Makefile
operand.h
output.c
plto_types.h
read-pe.c
sdkddkver.h
section.c
section.h
symbol.h
util.c
util.h
utilities.h
windef.h
winerror.h
winnt.h
x86-asm.h
x86-classes.h
x86-condition-codes.h
x86-dis.h
x86.h
x86-hash.h
x86-histogram.h
x86-icreate.h
x86-inline.h
x86-iprint.h
x86-jt.h
x86-liveness.h
x86-op.h
x86-prefix.h
x86-profdump.h
x86-registers.h
x86-schedule.h
x86-table-defs.h
x86-table.h
x86-util.h

<install_dir>/pe:
                        //windows portable executable (pe) specific code
cmdline.c
```

```
disasm
main.c
Makefile
read-pe.c
section.c

<install_dir>/psa:
                         //pointer set analysis code and data structures
abs_transformers.c
abs_transformers.h
aloc-list.c
Makefile
ptr_set_analysis.c

<install_dir>/util:
                         //some helpfule utilities
alloc.c
appmap.c
arraylist.c
hashtable.c
helper.c
linkedlist.c
Makefile
output.c
util.c

<install_dir>/x86:
                         //x86 architecture specific code
asm.c
dis.c
hash.c
histogram.c
icreate.c
inline.c
iprint.c
jt.c
liveness.c
Makefile
op.c
schedule.c
table.c
util.c
x86-condition-codes.h
x86-dis.h
x86-hash.h
x86-histogram.h
x86-icreate.h
x86-inline.h
x86-iprint.h
x86-jt.h
x86-op.h
x86-prefix.h
x86-profdump.h
```

```
x86-schedule.h
x86-table.h
```

APPENDIX 2. Disassembly Error Detection Tool

The tool for detecting disassembly errors consists of over 61,000 lines of C code for the tool that generates training inputs and over 3,000 lines of C code. It is not practical to attach this to this report; instead we attach the README file describing this tool. The software itself can be obtained from the PI (email: debray@cs.arizona.edu). Additionally, it uses C4.5, an open-source tool for constructing decision trees; however, this software must be obtained from its author (http://www.rulequest.com/Personal); restrictions in the license preclude us from distributing it.

```
Requirements
1. 64 bit processor
2. Udis 86 disassembler must be installed

Folder description
1. plto: Modified PLTO for generating training inputs
2. xtractFeatures: Code that generates feature vectors from PLTO output
3. C4.5: Decision Tree construction algorithm modified for the identifying
disassembly errors
4. scripts: Bash scripts that run the various processes

Steps to construct training data
1. Run the following commands on the folders:
      C4.5/R8/Src - make all
      plto - make
      xtractFeatures - make
2. Collect and feed the set of training executables that contain
relocation data to /plto/bin/plto
3. Pass the resultant set of files through xtractFeatures
(scripts/train_all.sh could be used with the appropriate folders provided)
4. Run scripts/removeCommentsAndGetC45Set.sh to obtain the training file
named allfeatures.data in the output folder
5. Create an appropriate allfeatures.names in the same output folder using
scripts/allfeatures.names as the base
6. Run scripts/train_all.sh to create the decision tree

Steps to construct test data and run tests
1. Collect the objdump output for the test file along with the section
header information
2. Create the feature set using scripts/make_testing_diff_features.sh
3. Run scripts/test_all.sh to get the results in the file scripts/Results
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* Jan 27, 2010	**2. REPORT TYPE** Final Performance Report	**3. DATES COVERED** *(From - To)* Dec 1, 2006 to Nov 30, 2009

4. TITLE AND SUBTITLE Topics in Low-Level Reverse Engineering, with Applications	**5a. CONTRACT NUMBER** FA9550-07-1-0019
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Saumya Debray	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Arizona P.O. Box 210077 Gould-Simpson #917 Tucson, AZ 85721-0077	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFOSR/NL 875 N RANDOLPH STREET SUITE 325, RM 3112 ARLINGTON VA 22203-1768	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-OSR-VA-TR-2012-0678

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution A

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of the project was to investigate automated techniques for analyzing computer malware codes, so as to simplify and accelerate the process of penetrating the defenses mounted by such malware to prevent analysis and extracting the internal logic of the malware. The investigators focused on analysis techniques that did not require the execution of the malware code. The project resulted in the development of a number of techniques for the analysis of executable files, including: a theoretical model for reasoning about malware code that modifies itself as it runs; an approach to automatically identify anti-analysis defenses in malware codes; an approach to automatically identify and emulate the code that performs the actual decryption of the malware code, and thereby extract the malware code; and an approach to detect possible errors in the instruction sequence obtained from examining a malware executable file. These results formed substantial components of one PhD dissertation and one MS thesis in Computer Science.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*